ECE 385 – Digital Systems Laboratory

Lecture 9 – Lab 6: SLC-3 Microprocessor

Zuofu Cheng

Spring 2018

Link to Course Website

ECE ILLINOIS

ILLINOIS

Common SystemVerilog mistakes

- Treating SystemVerilog as standard programming language
  - Languages like C and Java are designed for **sequential** computers
  - HDLs like Verilog & SV describe **digital hardware**
  - General hint, if you don't know what hardware (MUX, FF, adder, decoder, etc…) code you are writing describes, **you're doing it wrong**

```
always comb
    begin
        if a==b
            begin
                reg_8 my_reg(.*);
            end
        else
            ...
            end
        ...
    end
```

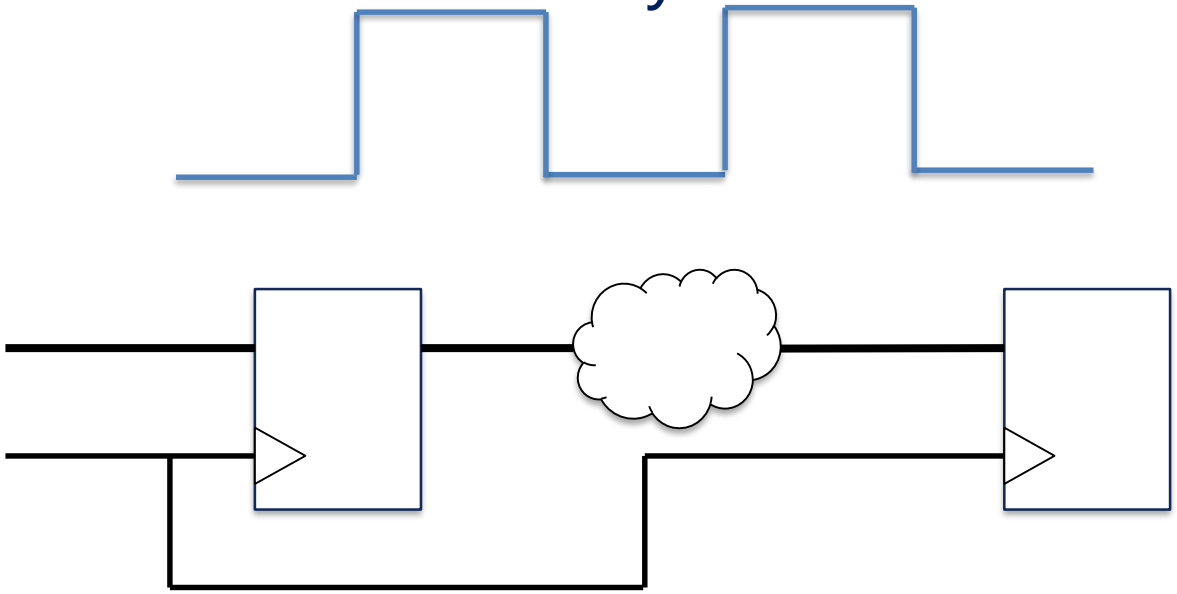
Common SystemVerilog mistakes

- What hardware is `current_state` describing? Is it consistent?
  - If you have errors, always ask yourself what hardware you are trying to describe with each variable (is it a connection, a register, a latch…)
  - If you don't know how to write a description for given hardware building block, look at examples or ask

```
enum logic [3:0] {A, B, C, D} current_state, next_state;
always comb
   begin
        unique case (current_state)
            A:  current_state = B;
            B:
            ...
        end
   endcase
```

Understanding Timing Model

- So far we haven't dealt with timing because designs are simple and we only run at 50 MHz ($\frac{1}{F_{max}} = T_{min} = 20\ ns$)
- More complex designs need to have timing constraints
- Need to understand the synchronous timing model



Constraint Files

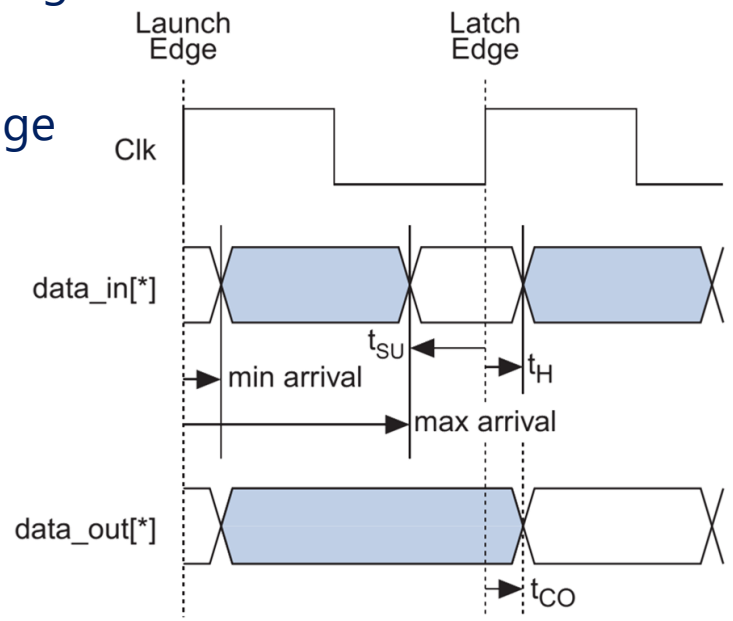
- FPGA place and route tool is iterative process
- The designer must tell the tool what is the minimum acceptable F_{max}
- Do this via .SDC file (Synopsys Design Constraint):
-

```
#**************************************************************
# Create Clock (where 'Clk' is the user-defined system clock name)
#**************************************************************
     create_clock -name {Clk} -period 20ns -waveform {0.000 5.000}
[get_ports {Clk}]

#creates a clock, applies it to all ports named "Clk" in toplevel
#note: -waveform specifies duty cycle, in this case 50%
```

- Place and route tool will keep optimizing until constraint (clk at least 50 MHz) is met or gives up
- Will need to do this for Lab 6! Or design might crash at 50 MHz

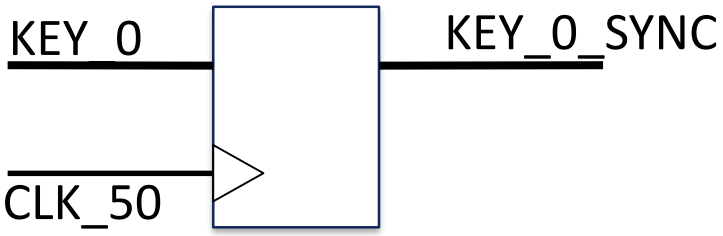
Non-ideal Flip-flop Behavior

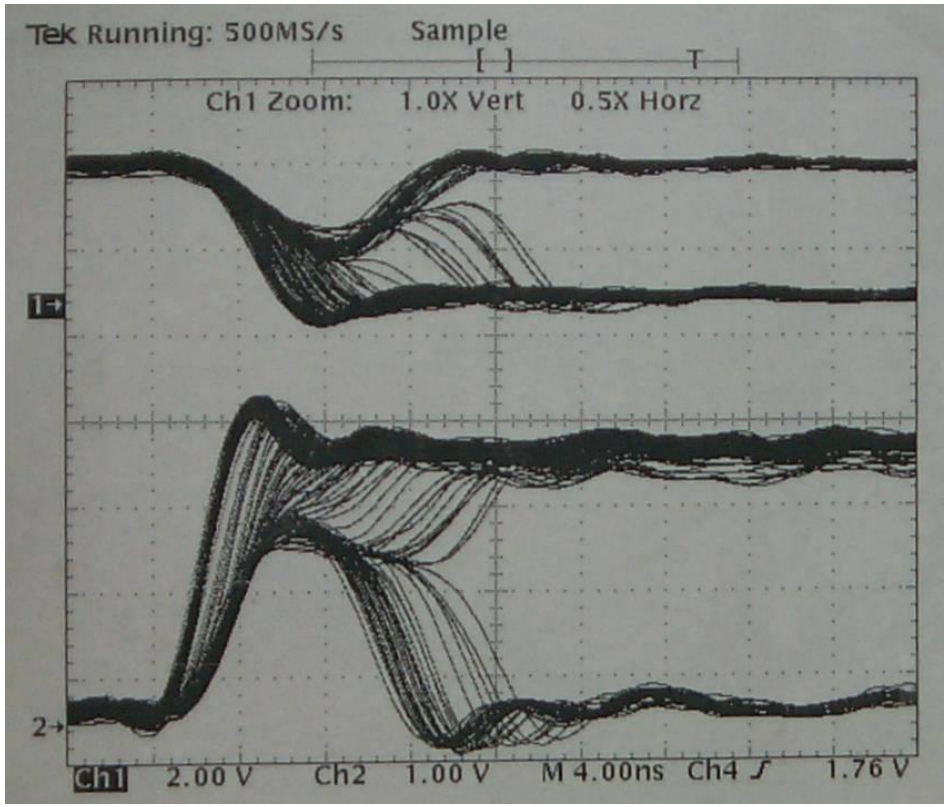
- Ideal flip-flop behavior, on clock edge, D->Q instantly
- In reality, flip flops do not behave ideally
- Non-ideal behavior characterized by 3 values: T_{CO}, T_{SU}, T_H
- Time from Clock to Output (T_{CO})
  - Time it takes for D to become Q after a clock edge
- Setup Time (T_{SU})
  - Time D needs to be correct/stable **before** clock edge
- Hold Time (T_H)
  - Time D needs to be correct/stable **after** clock edge
- Violation of any of setup and hold times
  - Causes potential meta-stability



Meta-Stability

- Violation of any of setup and hold times -> Causes potential meta-stability
- Four examples of meta-stable trajectories
- Note that this is a **different problem** than switch bounce (FPGA switches **are already** debounced)
- What are potential issues?
- Solution -> synchronizer





SystemVerilog Parameters

- Parameterized modules are useful to reduce code duplication
- Consider: 2-1 MUX – CPU may require widths of 2-1 MUX (2-bit, 8-bit, 16-bit)
- Can make separate modules MUX2to1_16, MUX2to1_8, but lots of redundant code (code looks exactly same)

```
module mux2
      #(parameter width = 8)
       (input  logic [width-1:0] d0, d1,
        input  logic s,
        output logic [width-1:0] y);

       always_comb begin
            if (s)
                  y = d1;
            else
                  y = d0;
       end
endmodule

//default with 8-bit width
mux2 my_8_mux (.d0(bus8_a), .d1(bus8_b), ...)

//16-bit mux instead of 8-bit
mux2 #(width = 16) my_16_mux (.d0(bus16_a), .d1(bus16_b), ...)
```

Lab 6: Goals (Week 1)

- Create SLC3 (Simplified LC3) microprocessor in SystemVerilog
  - 16-Bit Data Path
  - Memory-mapped I/O (only mapped peripheral is HEX displays using Mem2IO)
  - Register File (8 registers with control)
  - Other Registers
    - PC, IR, MAR, MDR, nzp status register
  - ALU and Memory Instructions
    - Add, Sub, Logical Ops, Load, Store
  - Control Flow instructions
    - Branch and Jump Subroutine
- Week 1: Demo only FETCH operation
  - Simulated and real memory
  - May use SV arithmetic operators (e.g. a = a + 1;)
  - Must pass timing and work at 50 MHz

Week 1 Demo

- Simulation of PC loading into MAR and PC incrementing. (1 point)
  - Use test_memory.sv
- Simulation of MDR loading into IR. (1 point)
  - Use test_memory.sv
- Correct FETCH operation on the board, showing IR on the hex displays.
  - Must use the physical memory (test_programs_image.ram) instead of the test memory (test_memory.sv). (1 point)
  - Mem2IO block takes up 4 HEX displays as I/O peripheral, use other 4 for displaying IR
  - Should halt after each FETCH so correct instruction can be seen on display
- **Even though demo is simple, plan on finishing at least data-path this week, or week 2's assignment will be impossible!**
  - Create all of the components in block diagram (register file, other registers, MUXes, ALU, branch logic, sign and zero extension blocks etc…) this week
  - Dedicate next week to control unit state machine (IDSU) and debugging

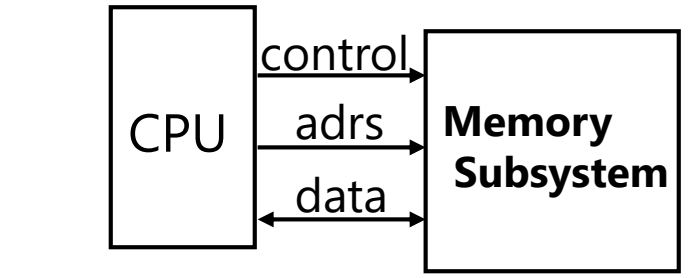
Available Documentation

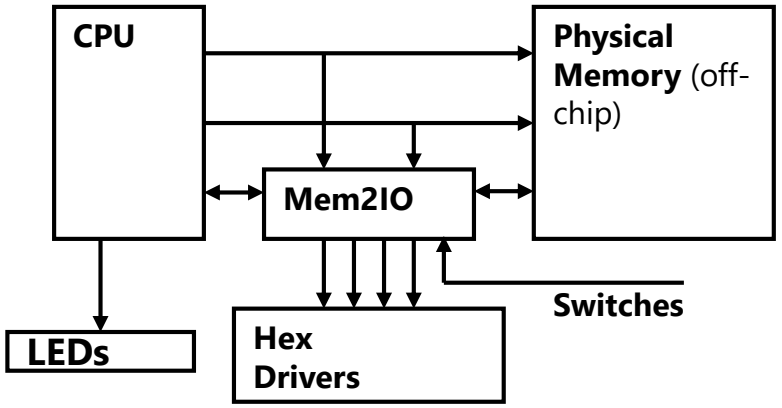
- Lab 6 Materials in Lab Manual
  - ISA breakdown (instruction coding for all 11 instructions)
  - Execution summary (RTL description for FETCH, DECODE, EXEC) for each instruction
  - Simplified block diagram
- Appendix C from P&P
  - Detailed ISA description of LC3
  - Full block diagram (with MUXes & individual registers)
  - Full state diagram
- Appendix A from P&P
  - Detailed programming guide for LC3
  - Explains instruction encoding and has examples for each instruction

Top Level Block Diagram

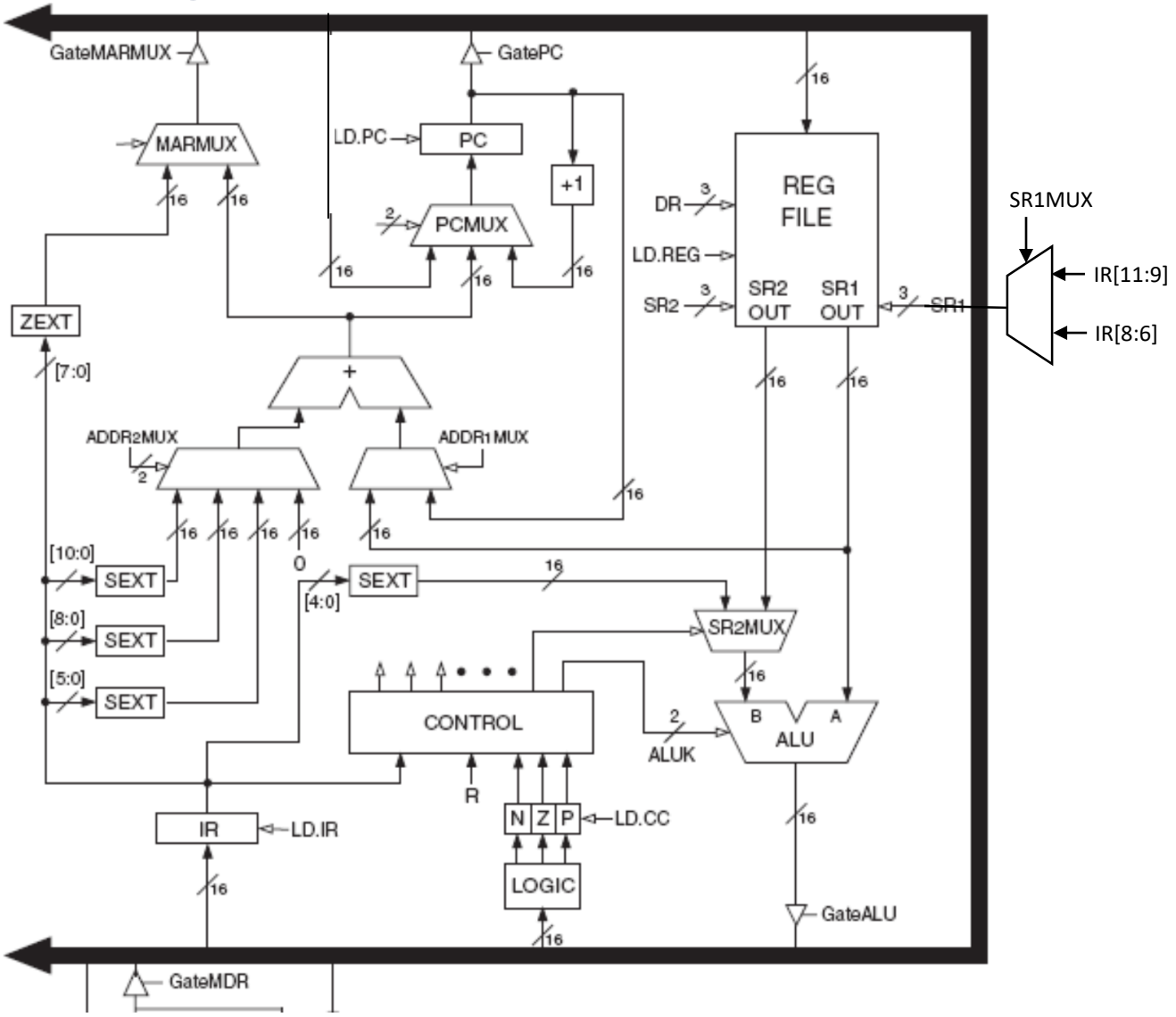
- Block diagram of FPGA top level
  - CPU only has control (r/w), address, data
  - I/O provided by Mem2IO block (I/O mapper)
  - Focus on control/addr/data signals

Physical I/O Device	Type	Memory Address	"Memory Contents"
DE2 Board Hex Display	Output	0xFFFF	Hex Display Data
DE2 Board Switches	Input	0xFFFF	Switches (15:0)





SLC-3 Block Diagram



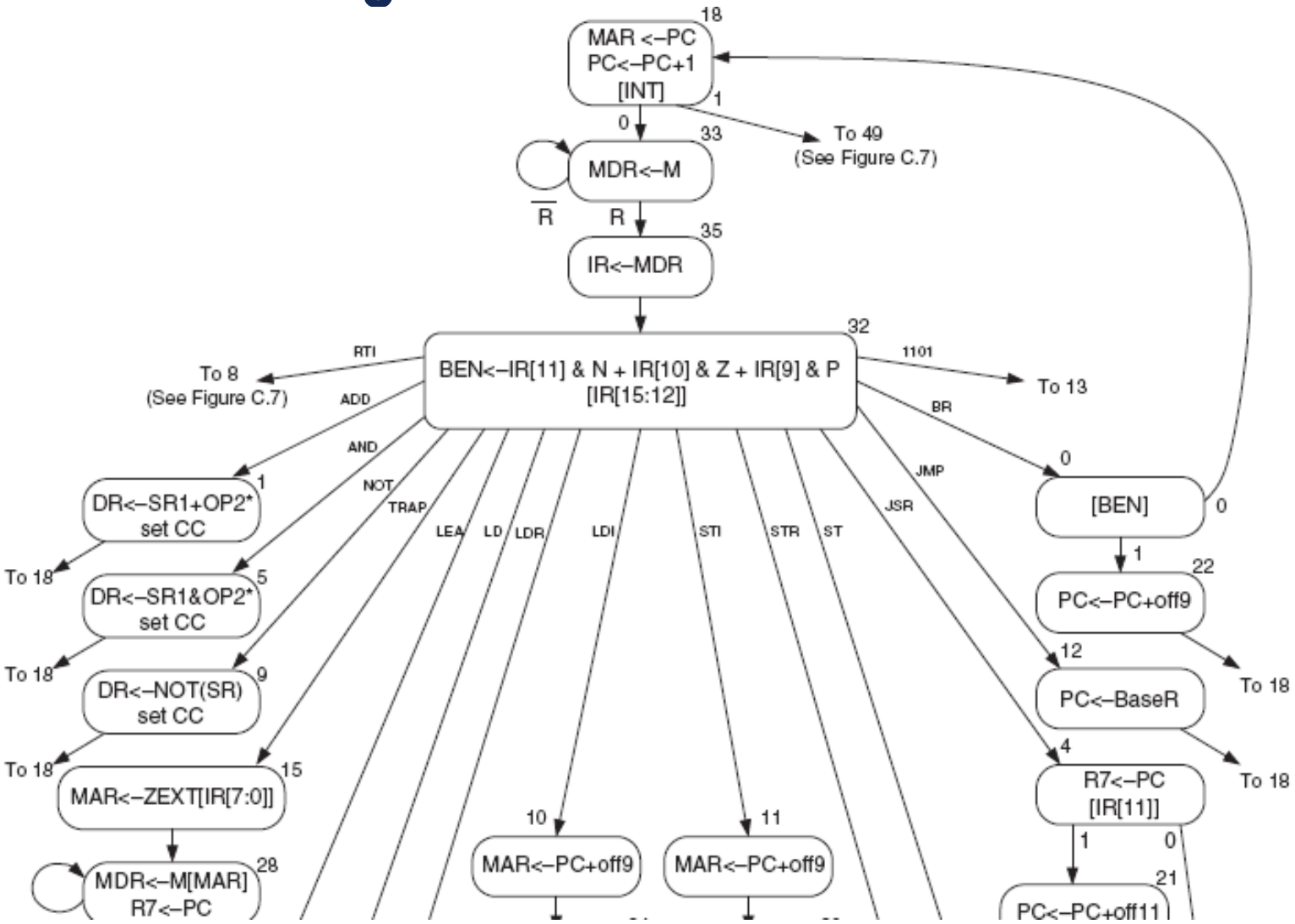
MDR is down here

For complete diagram check out the online materials

SLC-3 ISA – Subset of LC-3 ISA

Instruction	Instruction(15 downto 0)																Operation
ADD	0001			DR			SR1	0	00			SR2					R(DR) ← R(SR1) + R(SR2)
ADDi	0001			DR			SR	1	imm5								R(DR) ← R(SR) + SEXT(imm5)
AND	0101			DR			SR1	0	00			SR2					R(DR) ← R(SR1) AND R(SR2)
ANDi	0101			DR			SR	1	imm5								R(DR) ← R(SR) AND SEXT(imm5)
NOT	1001			DR			SR	111111									R(DR) ← NOT R(SR)
BR	0000			N	Z	P	PCoffset9										if ((nzp AND NZP) != 0) PC ← PC + 1 + SEXT(PCoffset9)
JMP	1100			000			BaseR	000000									PC ← R(BaseR)
JSR	0100			1	PCoffset11												R(7) ← PC + 1; PC ← PC + 1 + SEXT(PCoffset11)
LDR	0110			DR			BaseR	offset6									R(DR) ← M[R(BaseR) + SEXT(offset6)]
STR	0111			SR			BaseR	offset6									M[R(BaseR) + SEXT(offset6)] ← R(SR)
PAUSE	1101	ledVect12															LEDs ← ledVect12; Wait on Continue

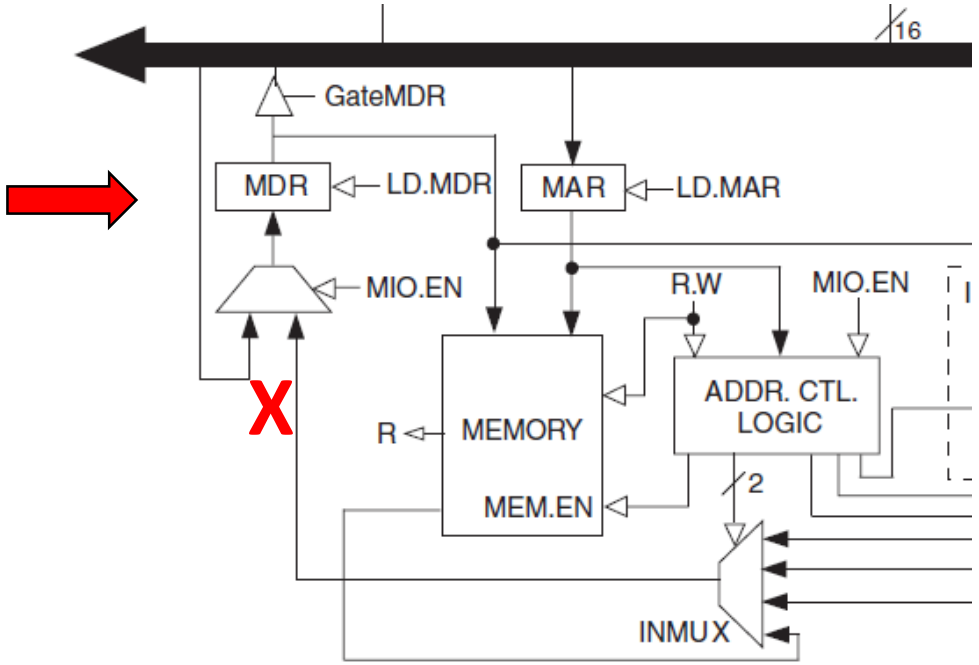
Partial State Diagram



For complete diagram check out the online mater

CPU to Mem2IO/SRAM Connection

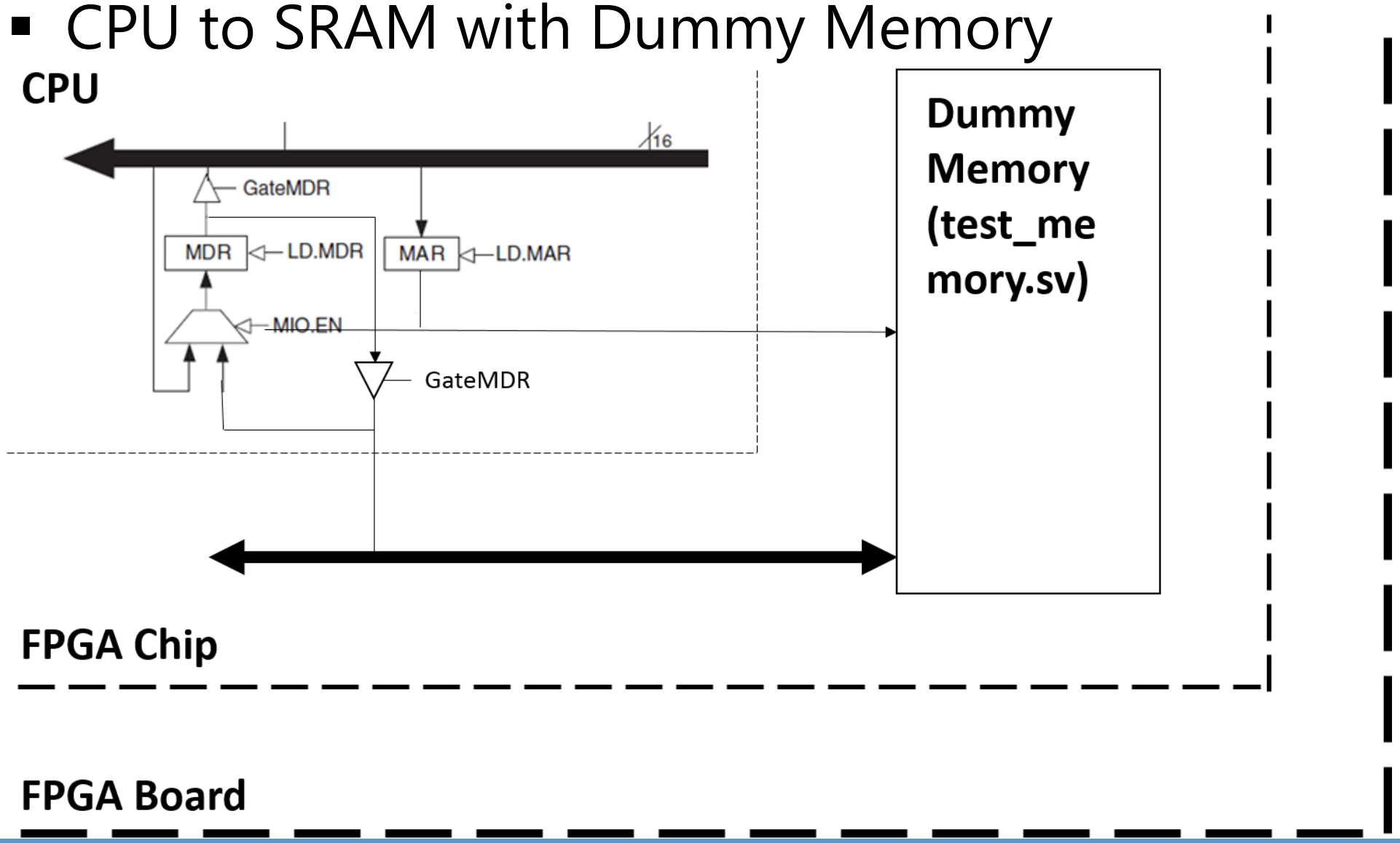
- **MEM2IO uses inout ports**
  - Appendix C has MDR directly connected to the memory.
    - **THIS IS NOT THE WAY**
  - We have to go through the tristate block, which converts uni-directional signals (internal to FPGA) to bidirectional signals



- MDR/MAR -> MEM2IO -> TRISTATE -> SRAM

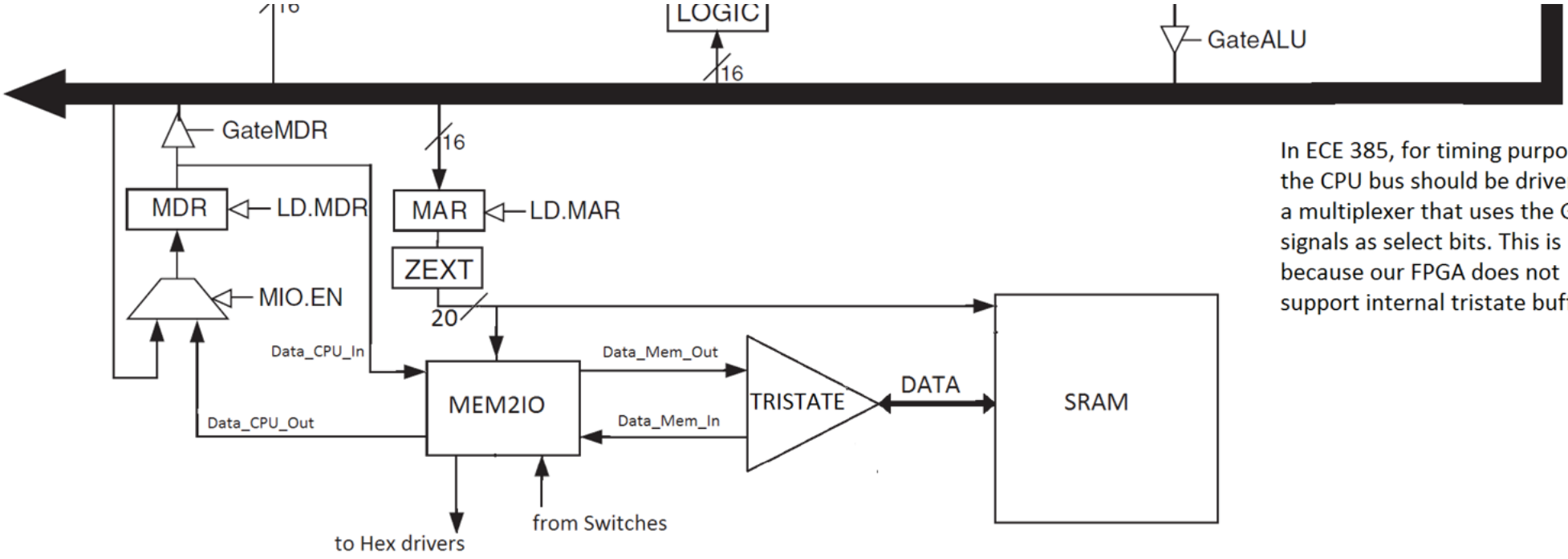
CPU to Dummy Memory Configuration

- CPU to SRAM with Dummy Memory



CPU to SRAM Configuration

- CPU to SRAM with Physical Memory



In ECE 385, for timing purposes, the CPU bus should be driven with a multiplexer that uses the Gate signals as select bits. This is because our FPGA does not support internal tristate buffers.

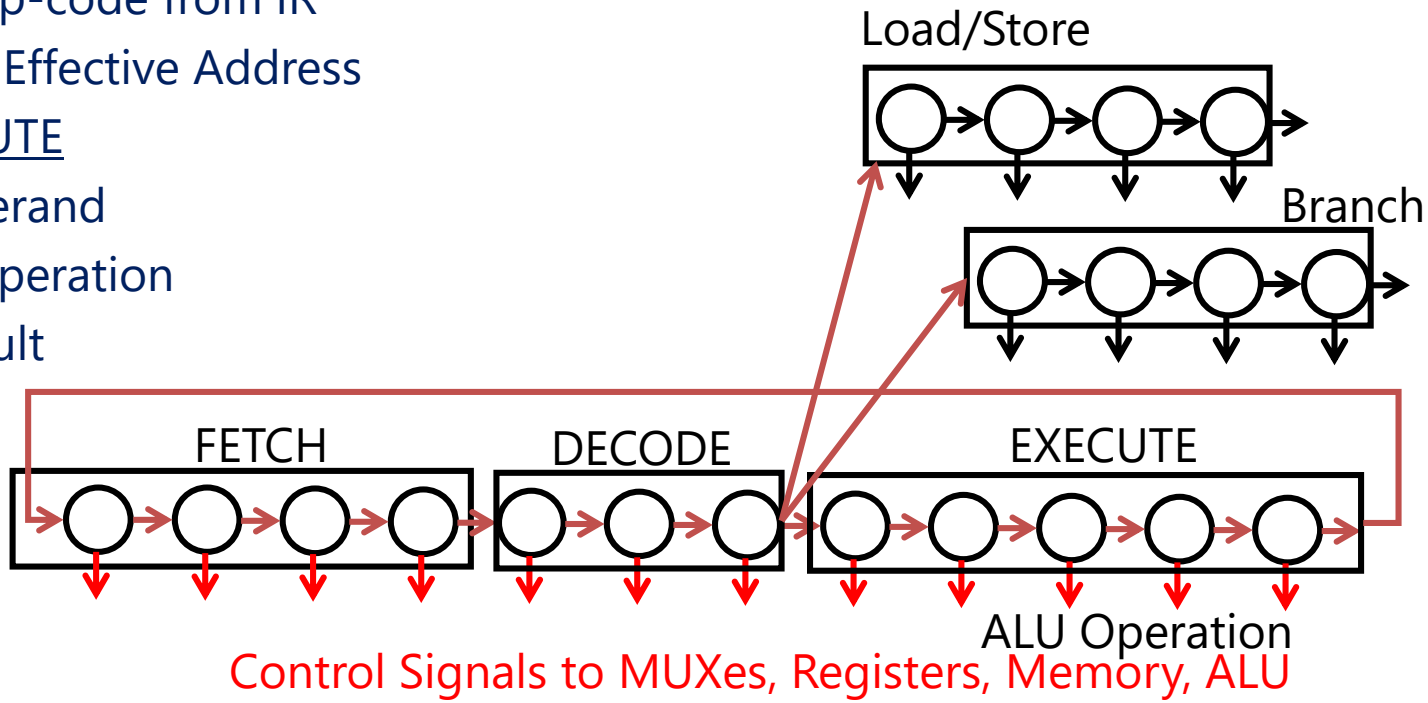
Instantiating Top Level Mem2IO & Tristate

- Mem2IO & Tristate blocks provided (top-level should have this in addition to ISDU - state machine)
- SRAM needs additional signals for operation (WE, OE, etc)
  - These signals should be driven by the ISDU in the appropriate states (when we need to read or write from SRAM)
- Remember:
  - MDR/MAR -> MEM2IO -> TRISTATE -> SRAM

Instruction Cycle

- Think of instruction cycle in three main phases
  - Phase1: FETCH
    - MAR <- PC;
    - IR <- Read Memory;
    - PC <- PC+1;
  - Phase2: DECODE
    - Decode op-code from IR
    - Compute Effective Address
  - Phase3: EXECUTE
    - Fetch Operand
    - Execute operation
    - Store Result

Note: Cycle counts in diagram not accurate



Understanding the Instruction Cycle

- My advice:
  - Print out ~10 copies of SLC block diagram
  - With different colored highlighter, trace out direction each MUX goes for each clock cycle in FETCH, DECODE and EXECUTE
  - FETCH should be common for all instructions
  - DECODE & EXECUTE will be different depending on instruction, start with a blank copy of block diagram and decide the DECODE & EXECUTE cycles for each instruction

FETCH Phase

- state1: MAR ← PC
- state2: MDR ← M(MAR); -- *assert Read Command on the RAM*
- state3: IR ← MDR;
  PC ← PC+1; -- "+1" inserts an incrementer/counter instead of an adder.
  Go to decode state – or halt (in the case of week 1)

<u>More details:</u>

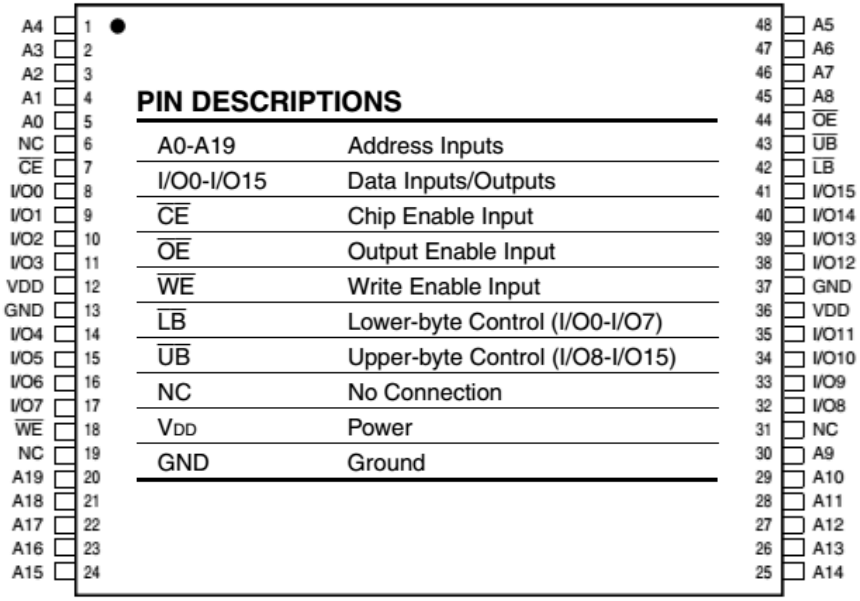
- MAR ← PC; MAR = memory address to read the instruction from
- MDR ← M(MAR); MDR = Instruction read from memory
- IR ← MDR; IR = Instruction to decode
- PC ← (PC + 1)

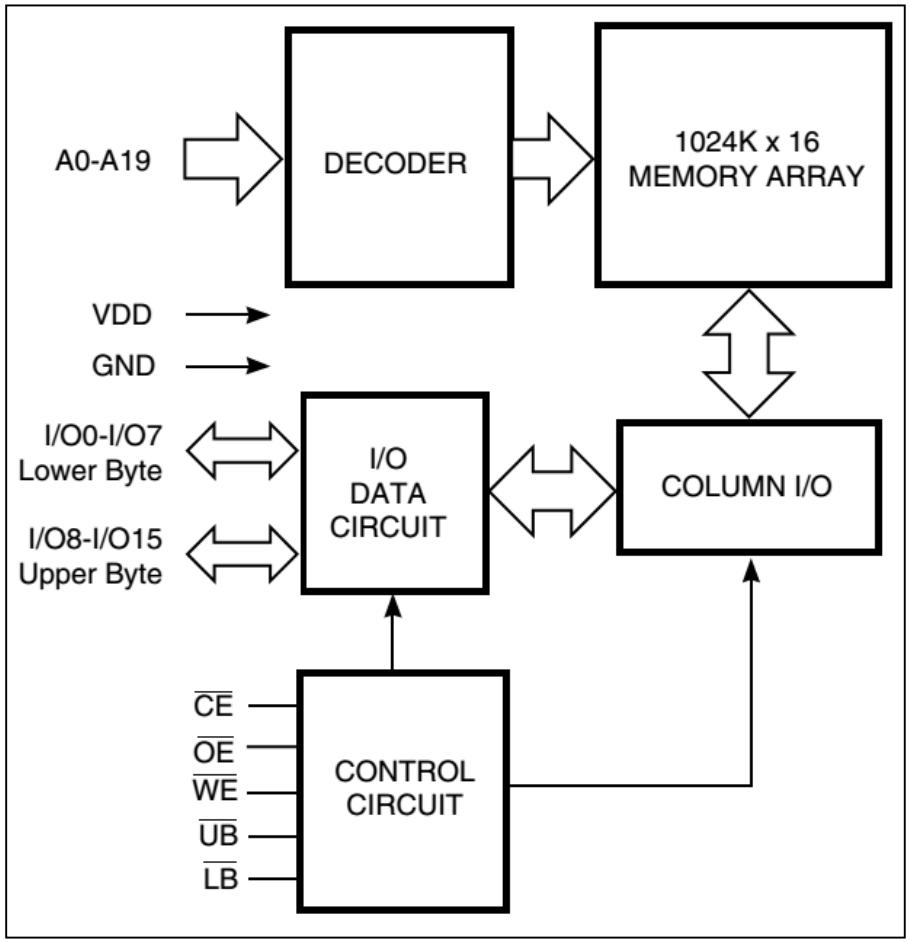
Provided IDSU Template

```
unique case (State)
…
S_33_2 : Next_state = S_35;              // Second cycle of mem FETCH (needed for SRAM)
S_35 :   Next_state = PauseIR1;          // Only for Week1
                                         // Bypass PauseIR in Week 2:
                                         // Next_state <= S_32;
PauseIR1 :              // Pause to display IR on HEX. (Week 1)
        if (~ContinueIR) Next_state = PauseIR1;
        else             Next_state = PauseIR2;
PauseIR2 :              // Wait for ContinueIR to be released. (Week 1)
         if (ContinueIR) Next_state = PauseIR2;
         else            Next_state = S_32 S_18; // Loop FETCH for Week

S_32 :
         case (Opcode) …

…
```

External SRAM

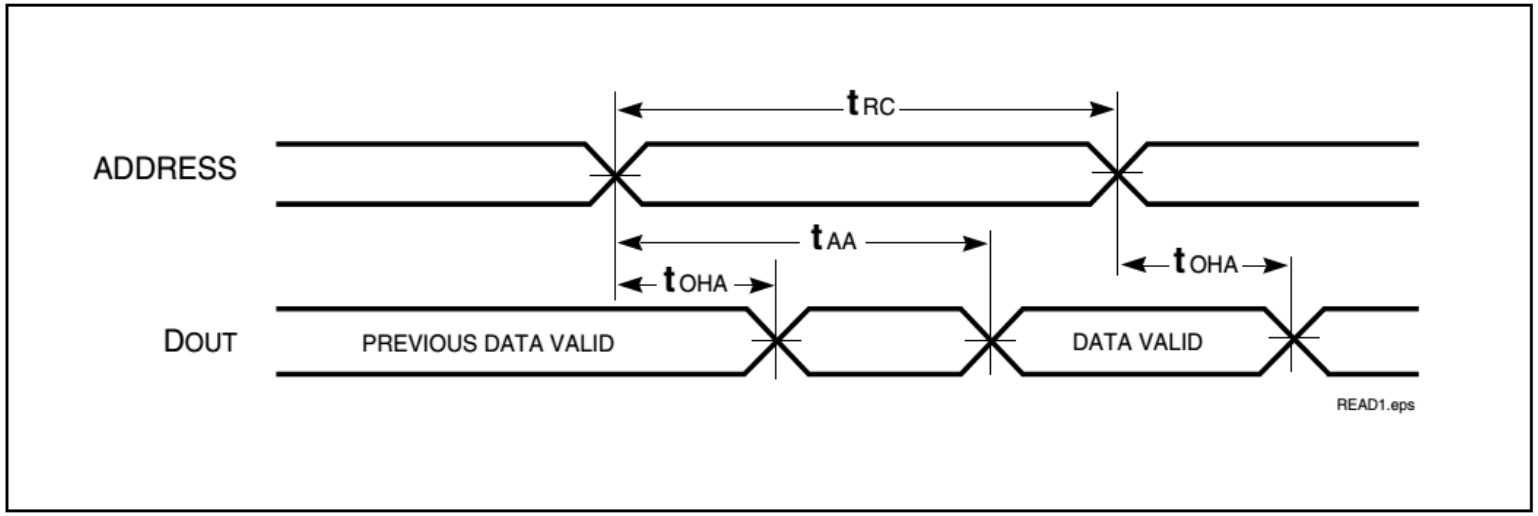
- 1M x 16 (2 Mbyte) organization
- Asynchronous (Access time = 10ns)
- 16 bit organization
- Byte access via UB/LB
- Datasheet here





External SRAM Timing (Read)

- External SRAM is **asynchronous**
- !CE = !OE = 0 (in diagram below)
- Data is valid 10ns after address is valid
- If CPU (and state machine) running at 50 Mhz, data guaranteed to be valid by next cycle (from address being valid)
- Tristate has internal flip flop for synchronization, so wait a total of 2 cycles in R



External SRAM Timing (Write)

- OE and WE have to be driven from your state machine
- OE and WE drive asynchronous SRAM, so they need to be synchronized!